

# The Portland Group Compiler Technology

STMicroelectronics

PGI Scalar and Parallel Compilers,  
Debuggers and Profilers for Hammer





# Overview

- PGI History
- Compilers, Debuggers, Profilers, Tools
  - Parallel Programming
  - Fortran, C & C++ Compilers
  - Debugger & Profiler
  - Cluster Development Kit
- Schedule
- Wrapup



# PGI History

- **1989** - Spin-off from Floating Point Systems (FPS Computing)
- **1990-93**
  - Vector, SW Pipelining, i860 Parallel C, C++, F77 compilers, Debugger, Profiler
  - **#1 in Performance and in Market-share (70%),** 20+ OEM vendors
- **1994-99**
  - Parallel PGHPF™ HPF compiler for most SMP and MPP systems
  - **#1 in Performance and Market-share (60%).** Cray, HP, Intel OEMs
- **1996-2000**
  - Parallel Pentium Pro C, C++, F77, Profiler for Linux, Solaris86, NT
  - Cluster Development Kit (CDK™) for Linux86
  - **#1 in Fortran and SMP Parallel Performance**
  - **#1 in Market-share of commercial Linux86 Fortran and CDK™**
- **Dec 2000** – Acquired by STMicroelectronics
  - Rebrand to **The Portland Group Compiler Technology**
  - DSP Architectures in addition to HPC
- **Jan 2001** – OpenMP & MPI Cluster Debugger & Profiler For Linux
- **Aug 2002** – **Collaboration w/ AMD for Hammer64 & Hammer32**



- AMD and STMicroelectronics are collaborating to bring The Portland Group Compiler Technology to x86-64
  - Cluster Development Kit® package being retargeted
  - CDK™ includes
    - F90, F77, C, C++ Scalar and OpenMP SMP parallel Compilers
      - Optimized 32-bit and 64-bit code generation
        - Focus on Fortran performance
    - Scalar & parallel PGDBG® debugger and PGPROF® post-mortem profiler
      - Multi-Threaded OpenMP, Multi-Process MPI Cluster and Hybrid Debugging and Profiling.
    - Linux and Windows, GUI
    - Turn-key installation, PBS, MPI-CH, Scalar/Parallel Math Libs, Training
  - Fortran Performance Goals
    - 32-bit code to outperform that of any other 32-bit compiler
    - 64-bit code performance within 95% of 32-bit code on same Hammer platform. Medium-term goal is to exceed 32-bit code performance.



# Good News

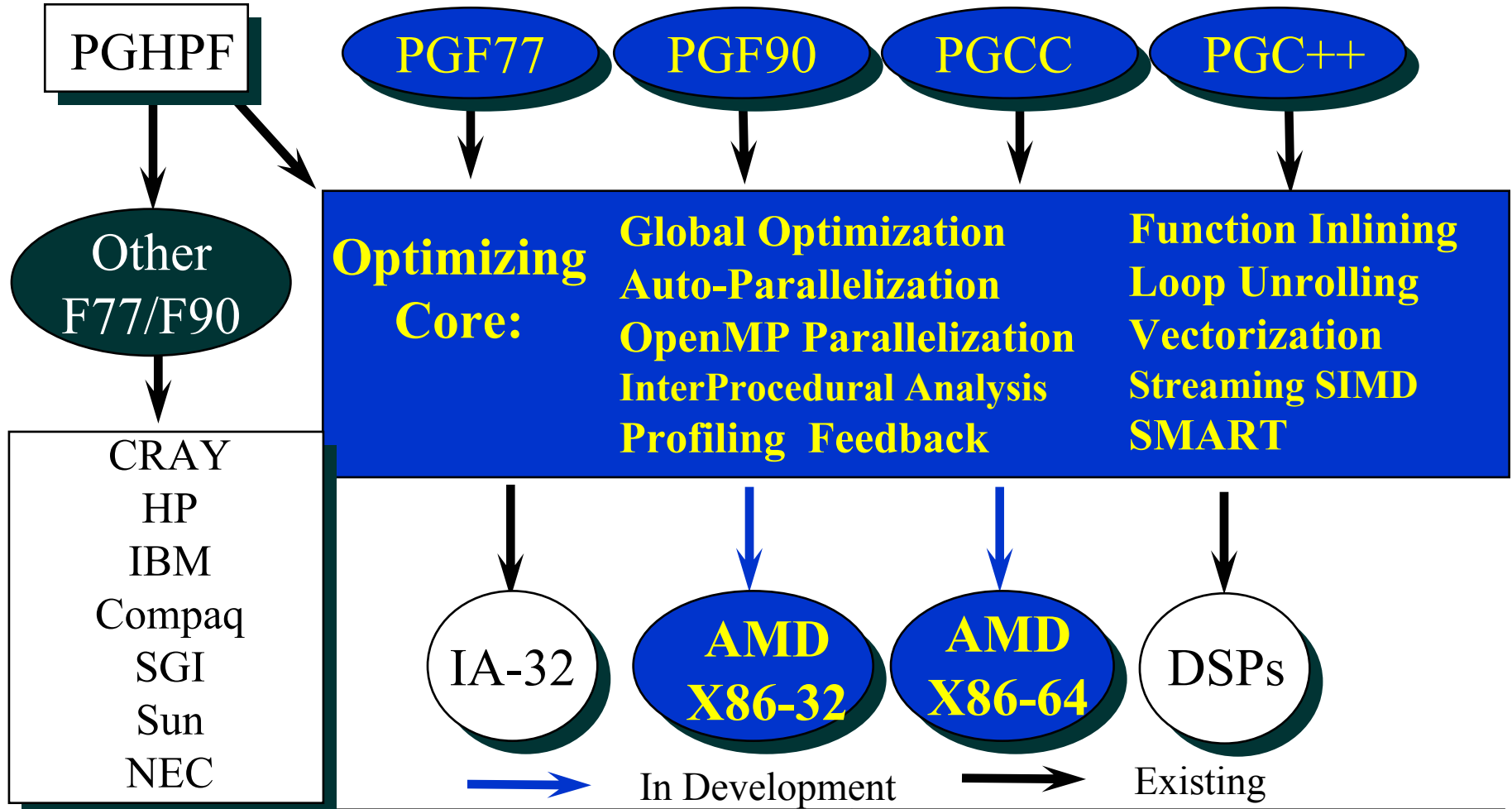
- **The good news is that the Hammer32 and Hammer64 architectures are simply advancements from the existing 32-bit x86 efforts that currently exist.**
  - Hammer improves in a number of ways
  - Isn't this what we all wanted?
- **As a result, PGI's effort to deliver world-class compiler performance to the Hammer32 and Hammer64 architectures for Linux and Windows will be very straight-forward.**
  - Very low risk – excellent leverage of existing technologies and products.
  - More registers and better ABI provide better opportunities to use more recent compiler technologies and to enhance compiler performance.
  - Compilers and tools will still require quite a bit of work to retarget, port and tune but the end result will be worth the effort!
- **Software Development Programming Model will not be changed**
  - **Assertion:** The SW Dev Programming Model for the Itanium relies too strongly on complex Programming Model (e.g. IPA, PFA, hand-tuning).
  - Hammer64 will not rely so much on these 'advanced' capabilities; however, they may be useful at times if used properly.



# Fortran, C, C++ Scalar & OpenMP Thread-Parallel Compilers



# The PGI Compilers





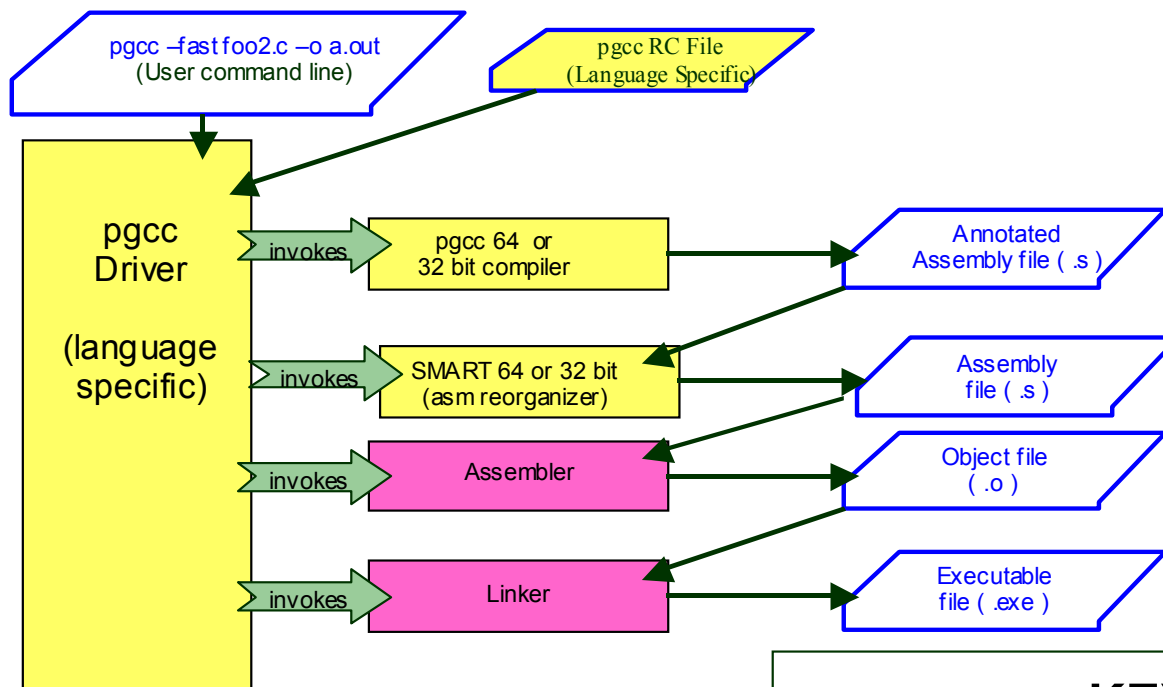
# Hammer64 Compilers

- 64-bit addressing
- 64-bit integer registers & operations
  - No longer software-simulated
- SSE/SSE2 for floating-point operations
  - Helps with porting applications from non-x86 systems, IEEE
  - X86 FP will be used w/ Hammer32 when necessary (ABI, Performance)
- More General Purpose & SSE/SSE2 registers
- Better ABI (sanity ruled)
  - e.g. reduced call overhead due to arg passing in registers
- Fortran data-types unaffected
  - INTEGER (32-bit), INTEGER\*8 (64-bit), F90 Kind parameters still the same, F90 REAL & DOUBLE PRECISION the same
- WIN64: long == 32-bits;    Linux64: long == 64 bits





## Tool Chain Flow Block Diagram



**NOTE:** If IPA or PFA is used, then a separate language specific IPA or PFA executable is invoked before the language specific compiler is invoked.

### KEY

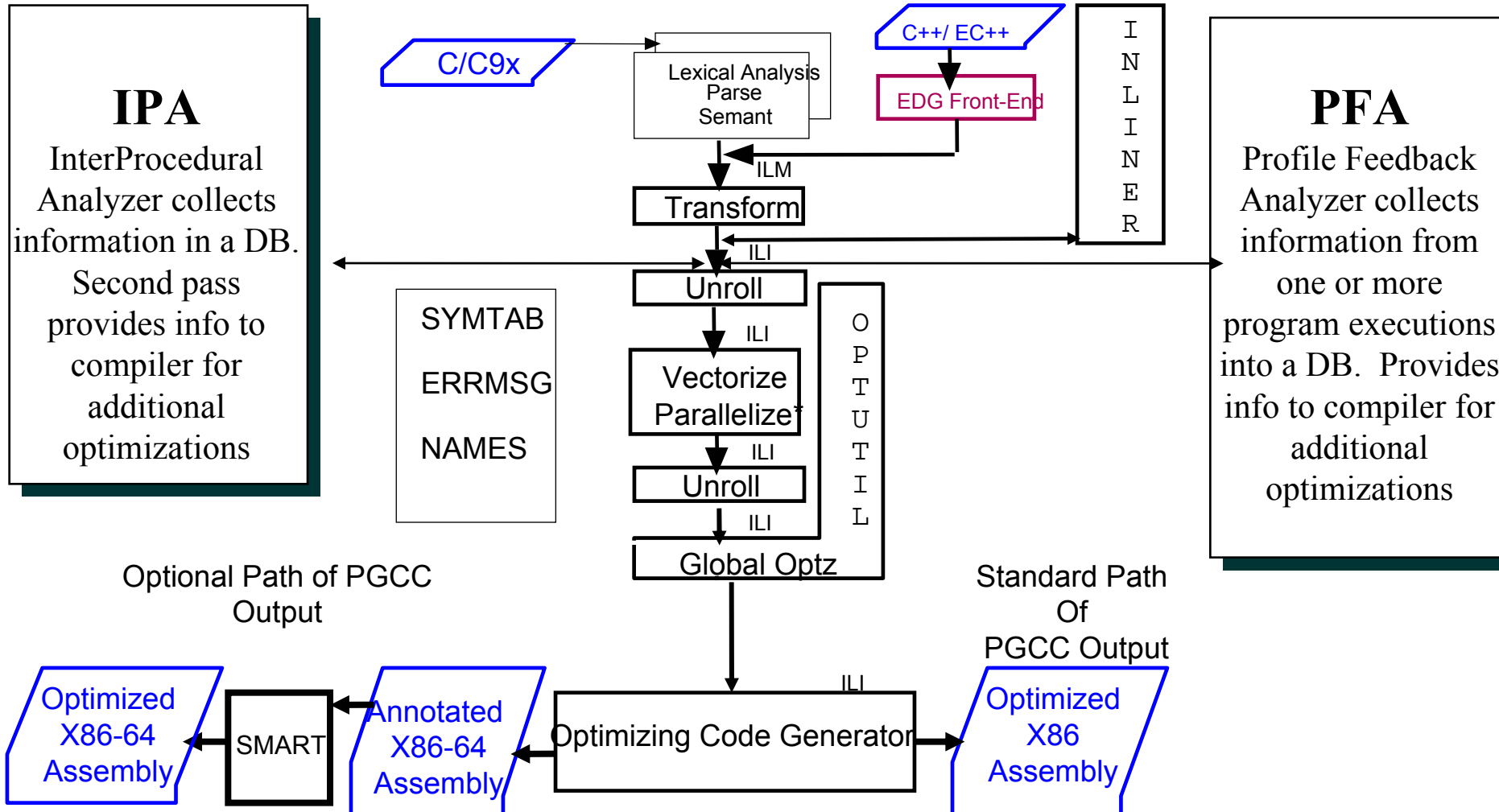
PGI supplied component

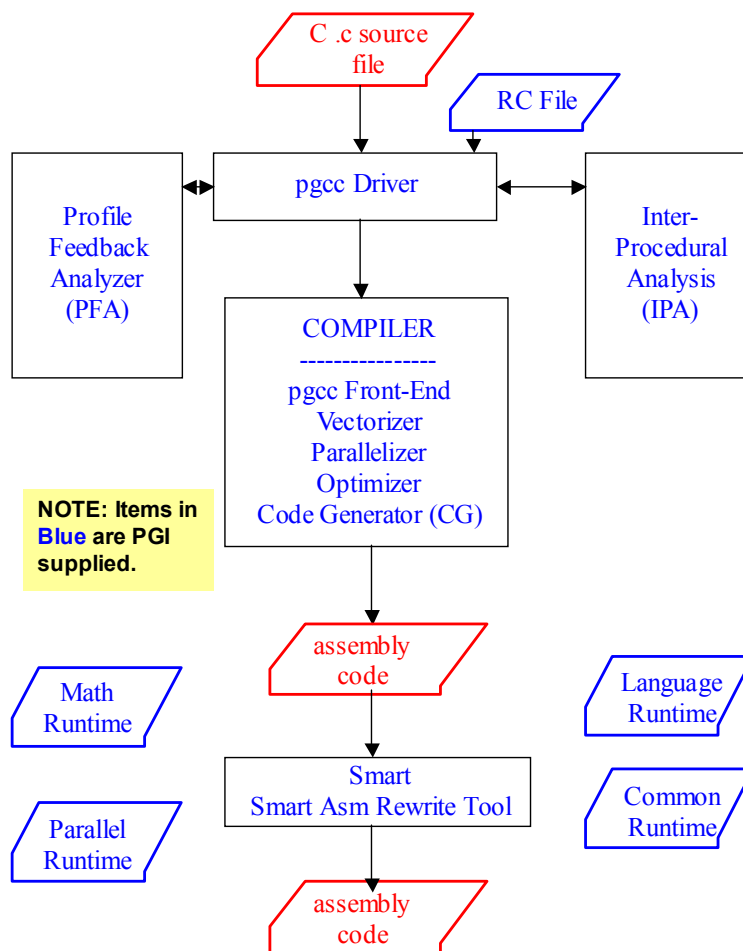
Input or Output file

Operating System supplied component



# PGCC Block Diagram







## Fortran/C/C++ Key Features

- PGF90® Fortran 90 Compiler
  - Robust, commercially-supported, F90 w/ F95 features
- PGC++ C++ Compiler
  - EDG 2.40 C++ support for ANSI C++
  - EDG Microsoft Compatibility Switch
  - STLport v4.5 Standard Template Library
- PGCC ANSI C Compiler w/ K&R support
- OpenMP & Auto-parallel for SMP Servers – F77, F90, C, C++
- Hammer64 specific – More Registers, Scalar SSE (No x87 FP), SMART Asm Optimizer, Tuned Intrinsics, Better ABI
- Hammer: SMART Asm Optimizer, SSE/SSE2, x86 FP when needed on x86-32, x86-32 ABI, IPA, PFA, Cache optimizations (e.g. tiling), Cache-aware Vectorizer
- Switches: -fast -fastsse -Mvect=fast -Mnontemporal -help -Minfo=all
- Miscellaneous Fortran: byte-swapping I/O & Large File support
- GNU Interoperability On Linux – cross-link *gcc/g77* objects/libraries



Candidate loops are vectorizable and utilize SSE instructions:

```
% pgcc -fastsse -Mvect=sse -Minfo matmul32.c
```

```
main:
```

```
    29, Interchange produces reordered loop nest: 30, 29
```

```
        Loop unrolled 10 times
```

```
    30, Outer loop unroll (4 times) and jam
```

```
    34, Interchange produces reordered loop nest: 35, 34
```

```
        Loop unrolled 10 times
```

```
    35, Outer loop unroll (4 times) and jam
```

```
    46, Outer loop unroll (4 times) and jam
```

```
    47, Loop unrolled 10 times
```

```
    53, Loop unrolled 10 times
```

```
    58, Call to __pgi_adotp4 generated
```

```
Linking:
```

```
%
```



## InterProcedural Analysis (IPA)

- **-Mipa=safe** when the whole call tree is not available to IPA, declares that unknown functions do not call back into the known functions, and do not change global variables
- **-Mipa=fast** common -Mipa options: -Mipa=const,globals,ptr,vestigial
- **-Mipa=const** propagates constants
- **-Mipa=ptr** pointer disambiguation
- **-Mipa=arg** removes arguments replaced by -Mipa=ptr,const
- **-Mipa=vestigial** eliminates functions that are not called
- **-Mipa=global** optimizes references to globals when not used in function call
- **-Mipa** equivalent to -Mipa=const



- MPI
  - *Low-level* library-based Fortran/C/C++ explicit parallel programming for *shared-memory*, *distributed-memory*, and *hybrid-memory* servers and clusters
- Automatic SMP
  - Compiler switches that direct the Fortran/C/C++ compiler to automatically detect loop-level parallel opportunities and create threaded parallel code. (SMP *shared-memory* servers only)
- OpenMP
  - *High-level* directive-based Fortran/C/C++ thread-parallel programming (SMP *shared-memory* servers only)
- HPF
  - *High-level* directive-based Fortran parallel programming for *shared-memory*, *distributed-memory*, and *hybrid-memory* servers & clusters



**PGI Options:**

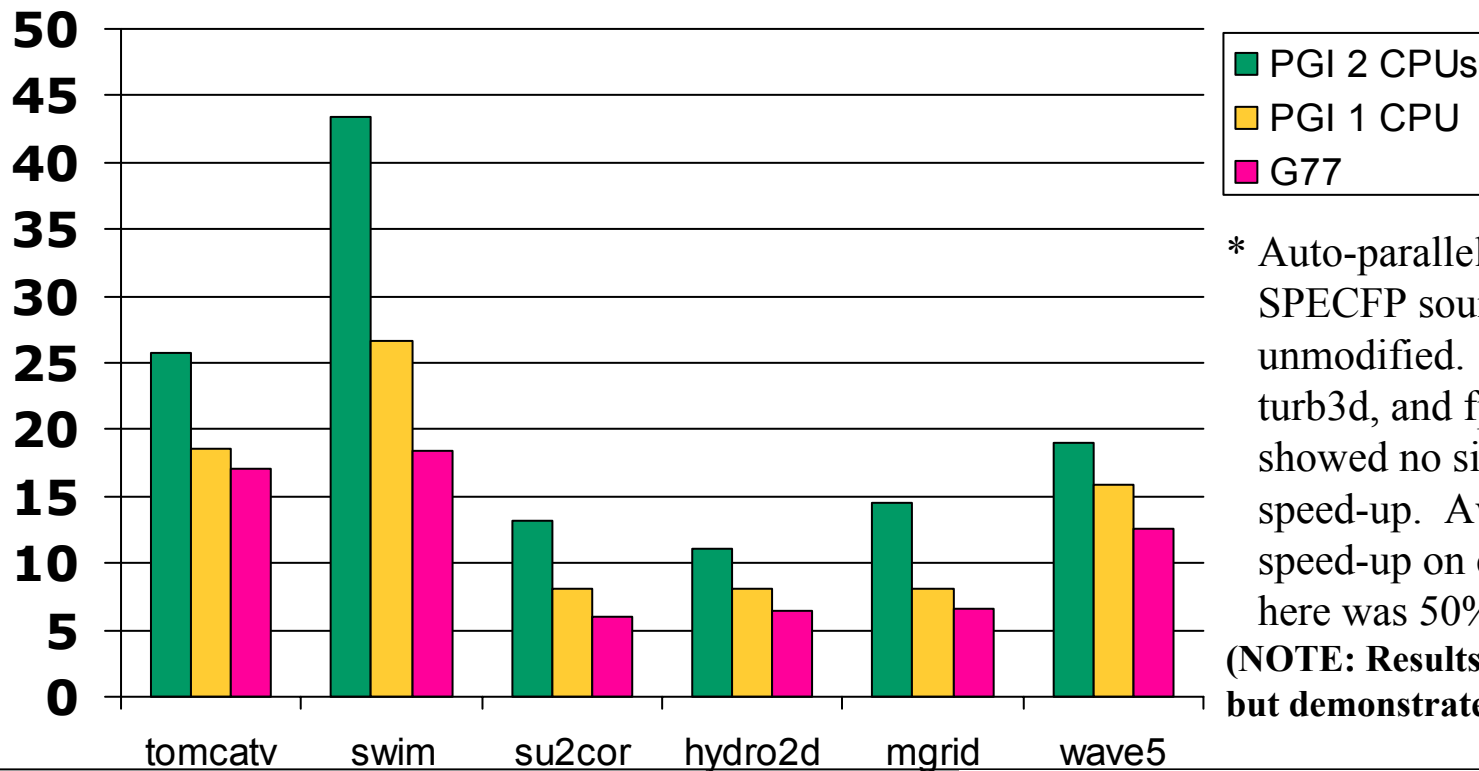
**-fast -Mconcur -Minline**

**G77/EGCS Options:**

**-O3 -fomit-frame-pointer -funroll-loops**

**-fstrength-reduce -fexpensive-optimizations**

**-ffast-math -malign-double**



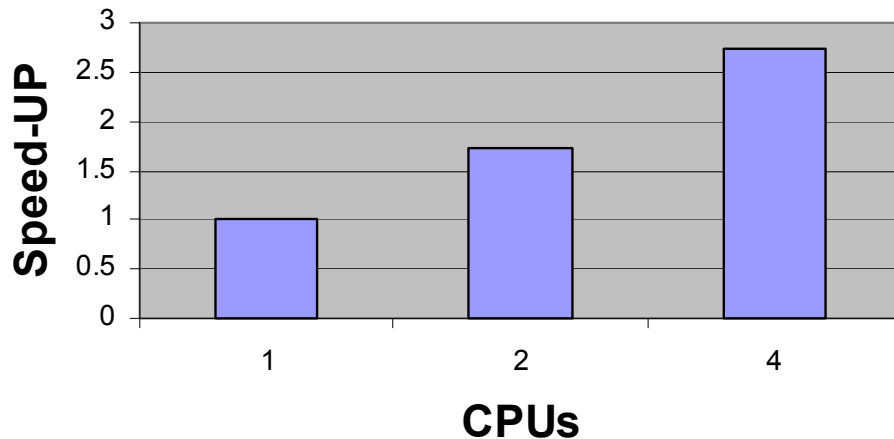
\* Auto-parallel only.  
SPECFP source completely unmodified. The applu, apsi, turb3d, and fpppp bmkrs showed no significant speed-up. Average parallel speed-up on codes shown here was 50% on 2 CPUs.  
**(NOTE: Results are quite dated but demonstrate auto-parallel.)**





## Example OpenMP Code

```
( 42)    for (l=1; l<=NTIMES; l++) {  
( 43) #pragma omp parallel private (j,i,ii,k)  
( 44)    {  
( 45) #pragma omp for  
( 46)        for (j=0; j<p; j++) {  
( 47)            for (i=0; i<m; i++) {  
( 48)                c[j][i] = 0.0;  
( 49)            }  
( 50)        }  
( 51)        for (i=0; i<m; i++) {  
( 52) #pragma omp for  
( 53)            for (ii=0; ii<n; ii++) {  
( 54)                arow[ii] = a[ii][i];  
( 55)            }  
( 56) #pragma omp for  
( 57)            for (j=0; j<p; j++) {  
( 58)                for (k=0; k<n; k++) {  
( 59)                    c[j][i] = c[j][i] + arow[k] * b[j][k];  
( 60)                } } }  
( 61)    (void) dummy(c);  
( 62)    }
```



## 111029 Elements

1 CPU Time = 119955 Seconds

2 CPU Time = 69113 Seconds

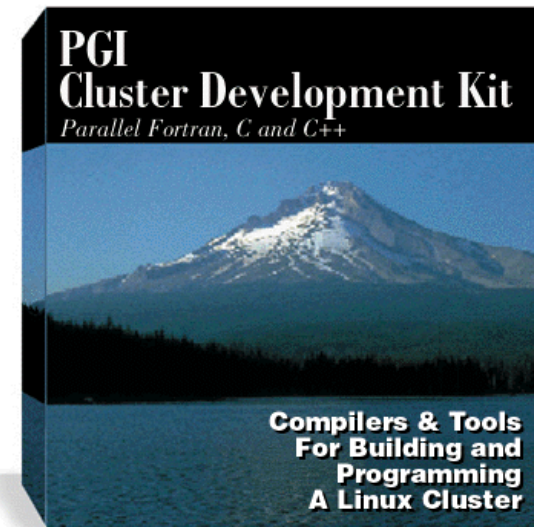
4 CPU Time = 43864 Seconds

**Quad 500 Mhz Xeon**

- **LSDYNA** = Over 750,000 lines of Fortran and C source code
- **1.74X Speedup** on 2 CPUs, **2.74X Speedup** on 4 CPUs
- Parallelized using *PGI Workstation* Fortran and **OpenMP**



# Debuggers Profilers Tools





- **Debugs Cluster-level MPI + OpenMP Thread-level Programs**
- **For each or all Processes AND for each or all Threads:**
  - Step, Break, Examine variables, Continue, ...
- **Shared-Memory Debugging**
  - OpenMP or Automatic Thread-level parallelism
  - For each OR for all Threads – Step, Break, Halt, Windows, Continue,...
- **Distributed-Memory MPI Debugging**
  - Debug MPI Processes
  - For each OR for all Processes – Step, Break, Halt, Windows, Continue,...
- **Hybrid-Memory MPI + OpenMP Debugging**
  - Multiple Nodes where each node has more than one SMP CPU
  - Control each process & control each thread separately or as groups.



## MPI Process Control

**Control  
MPI  
Processes**

The screenshot shows the PGDBG debugger interface. The main window displays a C program with MPI\_Init and a parallel loop. The PROCESSES pane shows process 1 selected. The PGDBG Active Threads and PGDBG Active Processes panes show the state of the running threads and processes.

**PGDBG Active Threads**

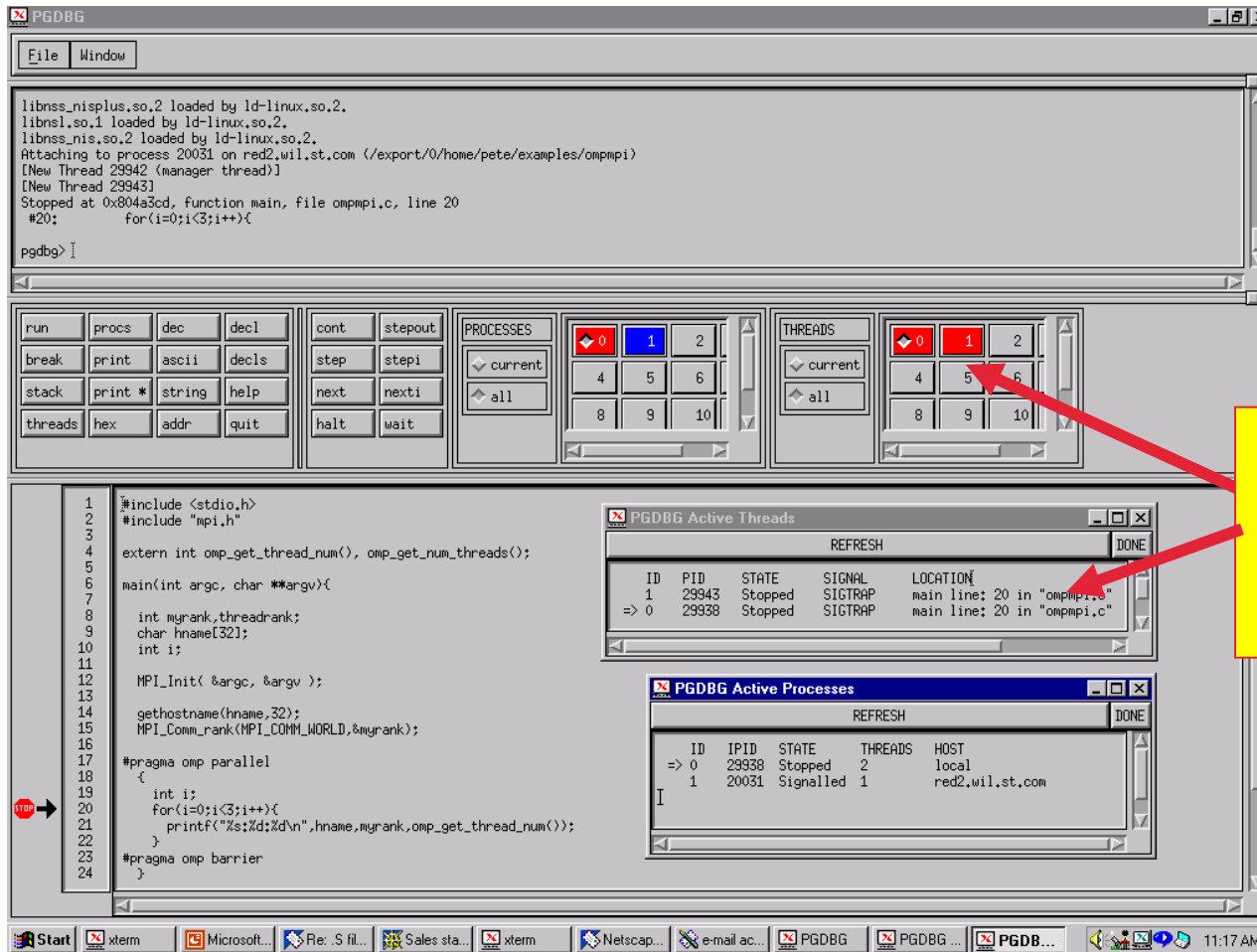
ID	PID	STATE	SIGNAL	LOCATION
1	29943	Stopped	SIGTRAP	main line: 20 in "ompmi.c"
=> 0	29938	Stopped	SIGTRAP	main line: 20 in "ompmi.c"

**PGDBG Active Processes**

ID	IPID	STATE	THREADS	HOST
=> 0	29938	Stopped	2	local
1	20031	Signalled	1	red2.wil.st.com



## Thread Control



The screenshot shows the PGDBG debugger interface. The main window displays a C program with OpenMP directives. The console window shows the program's output, including the number of threads created. The 'PGDBG Active Threads' window shows a list of threads, and the 'PGDBG Active Processes' window shows a list of processes.

**PGDBG Console Output:**

```
libnss_nisplus.so.2 loaded by ld-linux.so.2.
libnsl.so.1 loaded by ld-linux.so.2.
libnss_nis.so.2 loaded by ld-linux.so.2.
Attaching to process 20031 on red2.wil.st.com (/export/0/home/pete/examples/ompmapi)
[New Thread 29942 (manager thread)]
[New Thread 29943]
Stopped at 0x804a3cd, function main, file ompmpi.c, line 20
#20:      for(i=0;i<3;i++){
pgdbg> |
```

**PGDBG Active Threads:**

ID	PID	STATE	SIGNAL	LOCATION
1	29943	Stopped	SIGTRAP	main line: 20 in "ompmpi.c"
=> 0	29938	Stopped	SIGTRAP	main line: 20 in "ompmpi.c"

**PGDBG Active Processes:**

ID	IPID	STATE	THREADS	HOST
=> 0	29938	Stopped	2	local
1	20031	Signalled	1	red2.wil.st.com

**Control  
Threads**



libnss\_nisplus.so.2 loaded by ld-linux.so.2.  
libnsl.so.1 loaded by ld-linux.so.2.  
libnss\_nis.so.2 loaded by ld-linux.so.2.  
Attaching to process 20031 on red2.wil.st.com (/export/0/home/pete/examples/ompmi)  
[New Thread 29942 (manager thread)]  
[New Thread 29943]  
Stopped at 0x804a3cd, function main, file ompmpi.c, line 20  
#20: for(i=0;i<3;i++){  
pgdbg> |

run procs dec decl cont stepout procs step stepi stack print \* string help threads hex addr quit next nexti halt wait

PGDBG Active Threads

ID	PID	STATE	SIGNAL	LOCATION
1	29943	Stopped	SIGTRAP	main line: 20 in "ompmpi.c"
=> 0	29938	Stopped	SIGTRAP	main line: 20 in "ompmpi.c"

PGDBG Active Processes

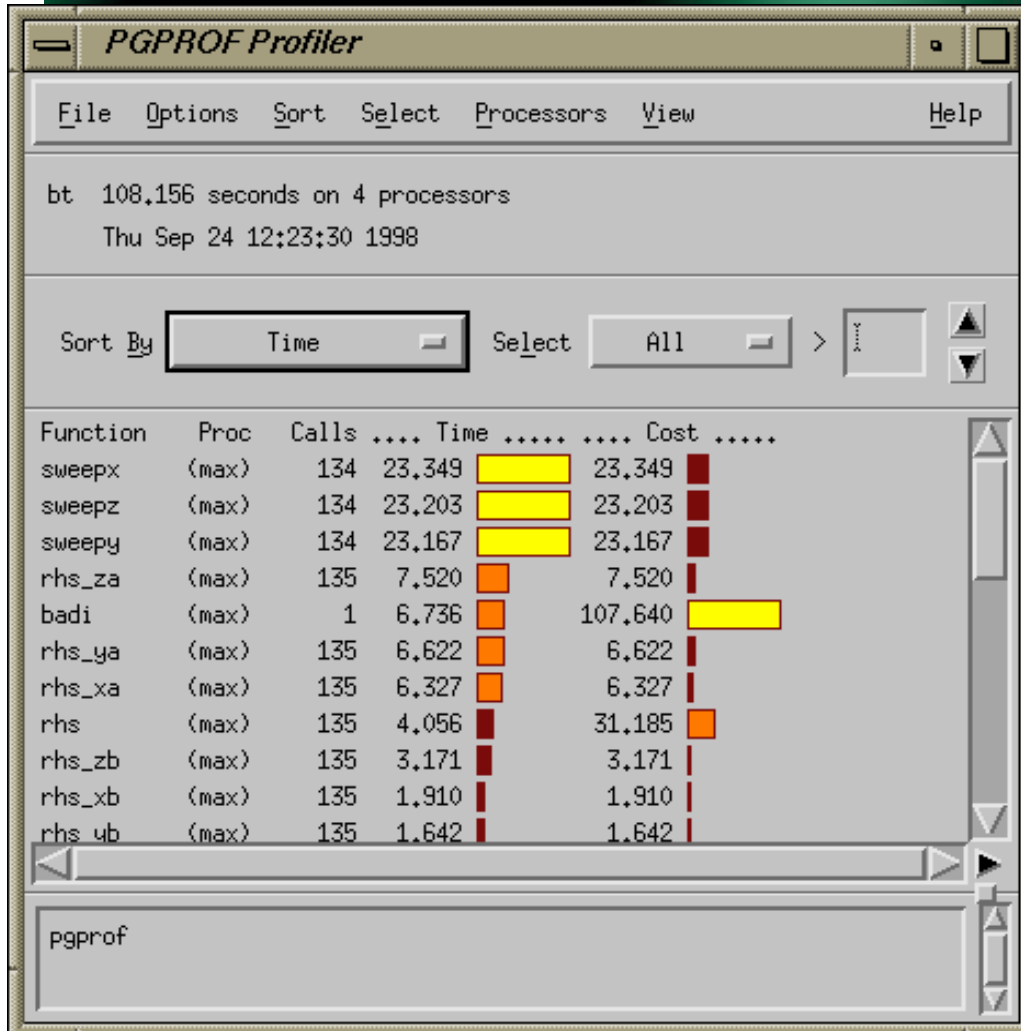
ID	IPID	STATE	THREADS	HOST
=> 0	29938	Stopped	2	local
1	20031	Signalled	1	red2.wil.st.com

**Step, Next,  
Break, Halt, Wait  
or Continue on  
Threads or  
Processes  
All or Individually**



- **Profile Cluster-level MPI + OpenMP Thread-level Programs**
- **For each or all Processes AND for each or all Threads:**
  - Examine Maximum, Minimum or Average Time spent in each or all.
  - Examine each process or thread individually
- **Shared-Memory Post-Mortem Profiling**
  - OpenMP or Automatic Thread-level parallelism
  - Examine Counts, Time or Cost on a Function Level or Statement Level
- **Distributed-Memory MPI Post-Mortem Profiling**
  - Profile MPI Processes and HPF Programs
  - Examine Counts, Time or Cost on Function Level or Statement Level
  - Examine Messages Sent and Messages Received
- **Hybrid-Memory MPI + OpenMP Profiling**





- *Cluster Profiling*
  - *Threads, Processes, Messages*
  - *OpenMP, MPI, HPF, Scalar*
  - *MPI Profiling*
    - *Send/Recv/Counts*
- *Post-Mortem*
- *Multi-Window*
- *Function Level*
- *Source Line Level*
- *C, some C++ & F90*
- *GUI & non-GUI*



## Parallel PGPROF

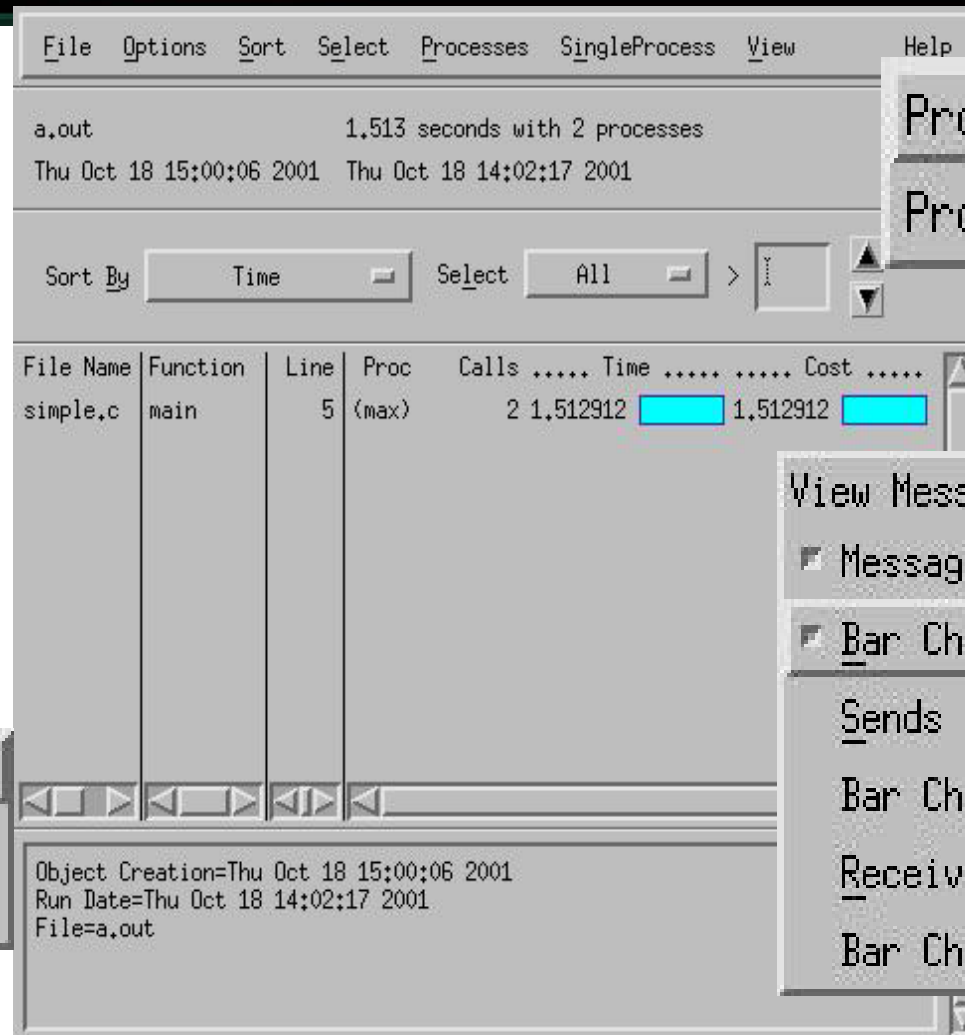
- **PGPROF**  
**postmortem**  
**parallel profiler**

- Profiles multiple processes!
- Profiles multiple threads!
- Profiles MPI messages!

Printer Options...

Help Options...

Source Directory...



File Options Sort Select Processes SingleProcess View Help

a.out 1.513 seconds with 2 processes  
Thu Oct 18 15:00:06 2001 Thu Oct 18 14:02:17 2001

Sort By Time Select All

File Name	Function	Line	Proc	Calls	Time	Cost
simple.c	main	5	(max)	2	1.512912	1.512912

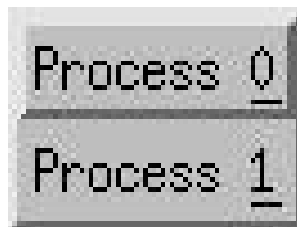
Object Creation=Thu Oct 18 15:00:06 2001  
Run Date=Thu Oct 18 14:02:17 2001  
File=a.out

Process 0  
Process 1

View Message Count  
Messages Total  
Bar Chart  
Sends  
Bar Chart  
Receives  
Bar Chart



- For **All** processes, observe:
  - Number of calls to each function
  - Time in each call
  - Total cost for each call
- OR select the processes individually



File Options Sort Select Processes SingleProcess View Help

a.out 1,513 seconds with 2 processes  
Thu Oct 18 15:00:06 2001 Thu Oct 18 14:02:17 2001

Sort By Time Select All >

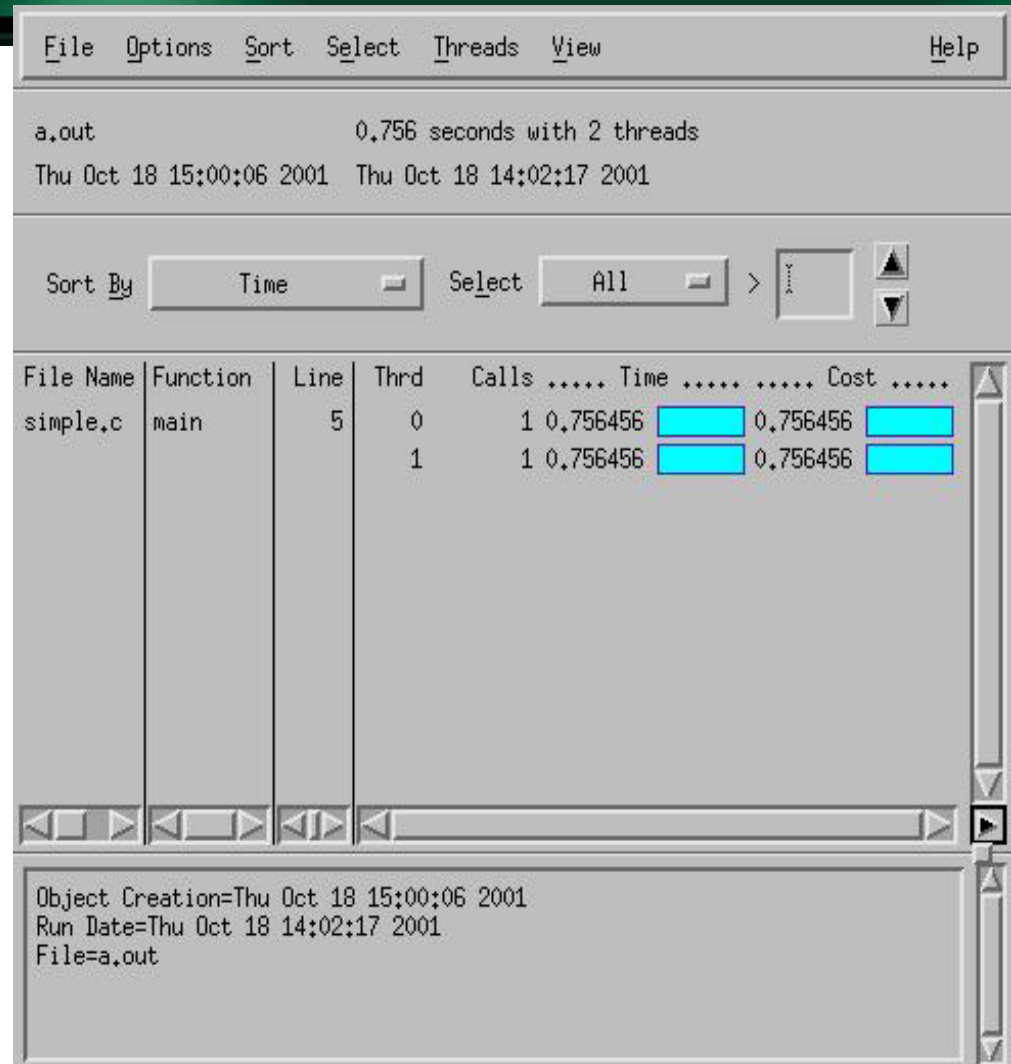
File Name	Function	Line	Proc	Calls	Time	Cost
simple.c	main	5	0	2	1,512912	1,512912
			1	2	0,117431	0,117431

Maximum  
Average  
Minimum  
Sum  
☒ All  
Individual...  
None

Object Creation=Thu Oct 18 15:00:06 2001  
Run Date=Thu Oct 18 14:02:17 2001  
File=a.out



- For each Thread (**Thrd**), observe:
  - Number of **Calls** to each function
  - **Time** in each call
  - Total **Cost** for each call
- **Time**
  - The amount of time spent within the function.
- **Cost**
  - The amount of time spent to execute the function and all children from that function.



The screenshot shows a window titled 'Function-level Thread Profile' with a menu bar (File, Options, Sort, Select, Threads, View, Help). The main area displays a table of function calls. The table has columns: File Name, Function, Line, Thrd, Calls, Time, and Cost. The data shows two calls to the 'main' function in 'simple.c' at line 5, one from thread 0 and one from thread 1, both taking 0.756456 seconds. The bottom of the window shows object creation and run date information.

File Name	Function	Line	Thrd	Calls	Time	Cost
simple.c	main	5	0	1	0.756456	0.756456
			1	1	0.756456	0.756456

Object Creation=Thu Oct 18 15:00:06 2001  
Run Date=Thu Oct 18 14:02:17 2001  
File=a.out



## Line-level Thread Profile

▪ **Within a function, at a line-level, observe for each thread:**

- Number of times each line was executed
- Time for each line
- Total cost for each line

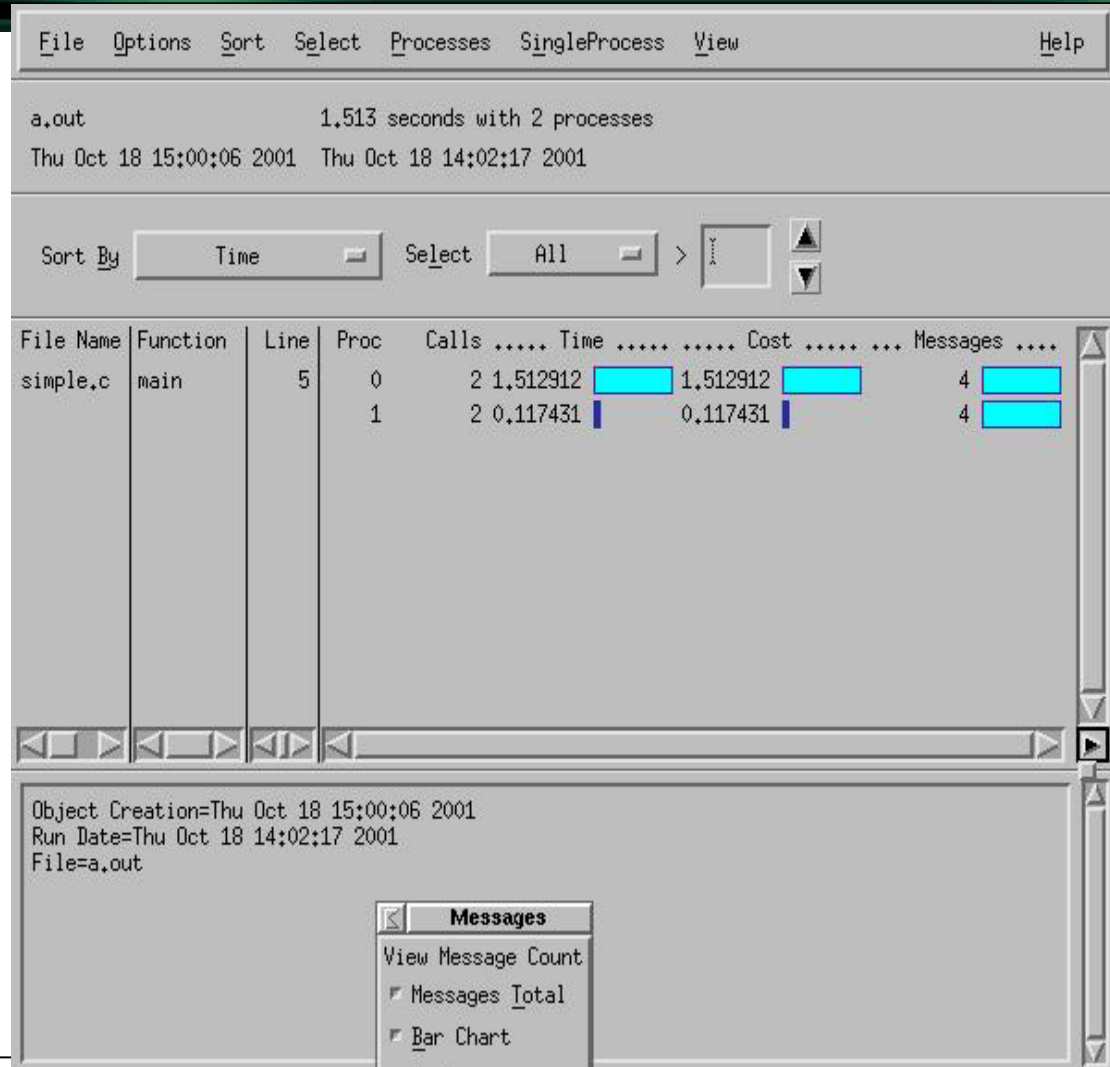
- ◢ All
- Calls
- Time
- Cost
- Coverage
- Executed
- Unexecuted

File Options Threads View Help						
main 0.756 seconds with 2 threads						
Thu Oct 18 15:00:06 2001 Thu Oct 18 14:02:17 2001						
Line	Source	Thrd	Count	Time	Cost	
16		1	1	1.30e-03	1.30e-03	
17	MPI_Init( &argc, &argv );	0	1	0.742272	0.742272	
		1	1	0.742272	0.742272	
18						
19	gethostname(hname,len);	0	1	0.000180	0.000180	
		1	1	0.000180	0.000180	
20						
21	MPI_Comm_rank(MPI_COMM_WORLD,&myrank);	0	1	3.14e-05	3.14e-05	
		1	1	3.14e-05	3.14e-05	
22						
Object Creation=Thu Oct 18 15:00:06 2001						
Run Date=Thu Oct 18 14:02:17 2001						
Source=simple.c						
Function=main						



## Function-level MPI Profile

- For each function and for each process, you can observe the number of MPI messages
- Determine the number of Sends
- Or of Receives





- ***MPI-CH*** - Pre-configured libraries and utilities for ethernet-based IA32/Linux clusters (<http://www-unix.mcs.anl.gov/mpi>)
- ***PBS*** – Portable Batch System batch-queueing from NASA Ames and MRJ Technologies (<http://pbs.mrj.com>)
- ***ScaLAPACK*** - Pre-compiled distributed-memory parallel Math Library
- ***Training*** – Tutorials (OSC), exercises, examples and benchmarks for MPI, OpenMP and HPF programming





- Configured by default as a single space-shared queue – PBS is very configurable! Space-shared and time-shared queuing options available, custom scheduling available
- Option of having a dedicated Front-end Node or using one of the Compute Nodes to submit/monitor jobs
- Supports both interactive and batch jobs
- Complies with POSIX 1003.2d standard for shells, utilities, and batch environments
- Available on many platforms besides Linux – PBS is gaining popularity as the de facto standard open source Queuing utility
- Commercial support available from MRJ Technologies





**Submit Job Dialog**

**SCRIPT** Prefix **#PBS**

**FILE..**  **load** **save**

**OPTIONS**

**Job Name**  **Priority**

**Account Name**  ☐ **Hold Job**

**Destination**

**When to Queue** ☒ **NOW** ☐ **LATER at..**

**Notify**  **when** ☒ **job aborts**  
☐ **job begins execution**  
☒ **job terminates**

**OTHER OPTIONS**

**concurrency set..**  
**after depend..**  
**before depend..**  
**file staging..**  
**misc..**

**Output**  
☒ **Merge to Stdout**  
☐ **Merge to Stderr**  
☒ **Don't Merge**

**Retain**  
☒ **Stdout in exec\_host:<jobname>.o<seq>**  
☒ **Stderr in exec\_host:<jobname>.e<seq>**

**Stdout File Name..**  **on hostname:**

**Stderr File Name..**  **on hostname:**

**Resource List** **help-unicos8**

resource	value
ncpus	<input type="text"/>

**Resources**

nodes	value
2	<input type="text"/>

**Environment Variables to Export** ☐ **Current**

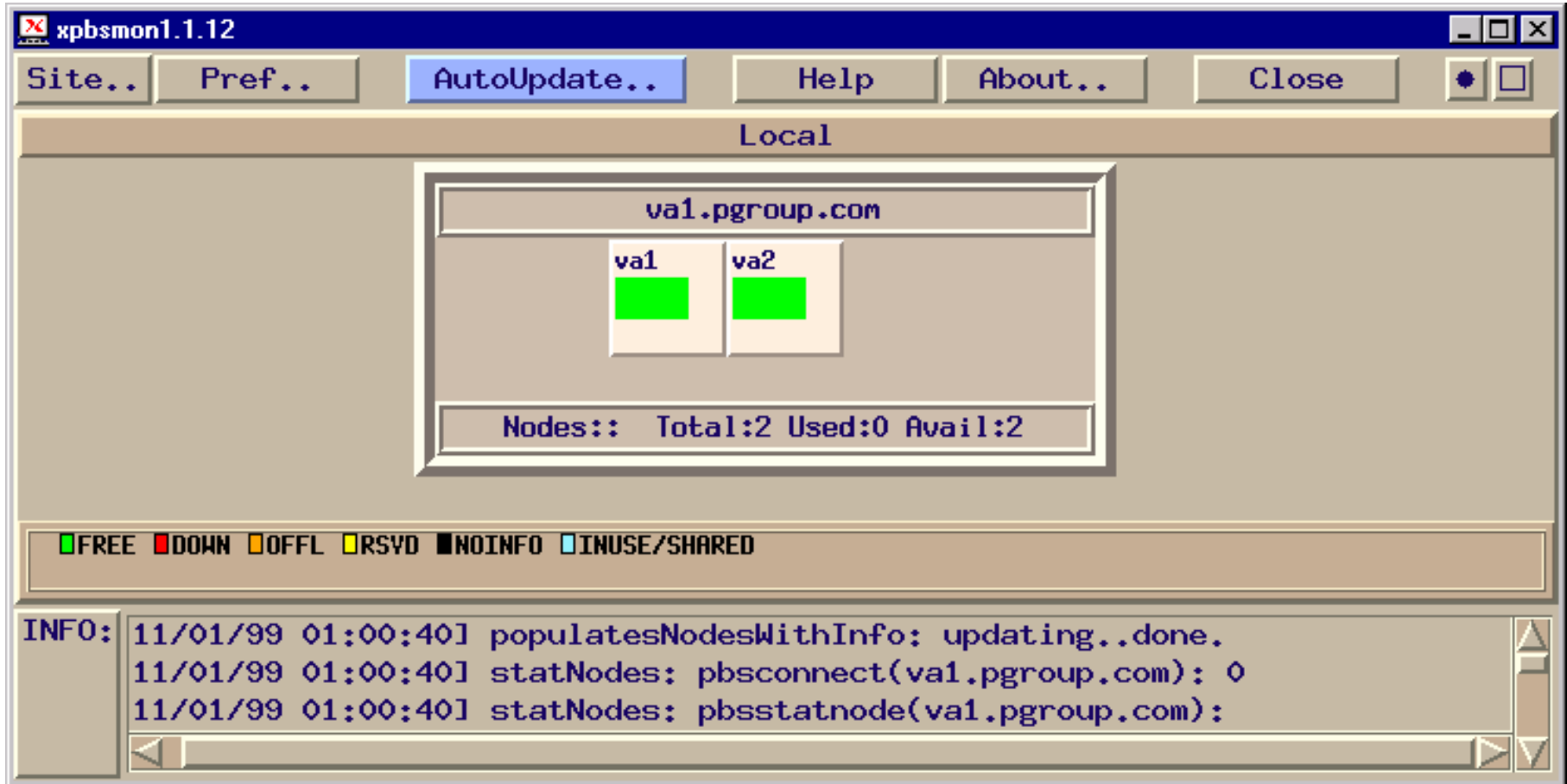
variable	value
<input type="text"/>	<input type="text"/>

**Variables**

variable	value
<input type="text"/>	<input type="text"/>

**confirm submit** **interactive** **cancel** **reset options to default** **help**

## XPBS – Job Submission GUI





# Schedule



# Schedule

## ■ Linux64

- Beta versions of C, Fortran, C++ compilers prior to Hammer launch
- Commercial Compiler Release in 1H03
- Commercial Parallel Debugger/Profiler Release in Q4 03
- Current Effort is On track

## ■ Windows64 Goals

- Beta versions of C, Fortran, C++ compilers prior to Win64 commercial release
- Commercial Compilers in conjunction with Win64 commercial release
- Commercial Compilers in MingW Win32 environment
- Commercial Parallel Debugger/Profiler about 3-6 months after compilers



AMD, the AMD Arrow Logo and combinations thereof are trademarks of Advanced Micro Devices, Inc.

ST, the ST Logo, The Portland Group and The Portland Group Compiler Technology Logo are trademarks of STMicroelectronics

Other product names used in this presentation are for identification purposes only and may be trademarks of their respective companies.